View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Elsevier - Publisher Connector

Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 160 (2006) 211–224

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Component-Based Specification of Distributed Systems

Grant Malcolm[1]

Department of Computer Science
University of Liverpool
UK

Abstract

We suggest that *hidden algebra* can provide a setting for component specification and composition that has the advantages of algebraic specification, without the disadvantages of object-oriented approaches where communication between components is mediated solely by method invocation. We propose a basic composition mechanism for hidden algebraic component specifications that is based on communication through shared subcomponents, and show that this composition mechanism on specifications extends naturally to allow models (or implementations) of the component specifications to be amalgamated into a model of the composite system.

Keywords: Component-based software, algebraic specification, hidden algebra, distributed systems, amalgamation.

1 Introduction

As part of a general trend towards decentralisation [16], computer systems tend more and more to be constructed from distributed, self-contained, and possibly autonomous units. The challenges that this poses to computer science are reflected in the growth of new paradigms such as component-based, service-oriented, and aspect-oriented software, and of new languages for modelling, specifying, composing, and co-ordinating these units. The object paradigm certainly helped set these developments moving: code could be organised at the level of classes, architectures at the level of class instances, and both these levels could be seen as comprising self-contained, even autonomous units. Distributedness is an essential part of the object paradigm, with interaction between instances being mediated by (possibly remote) method invocation, and type systems that include interfaces and abstract classes allow systems of interacting objects to be built from subsystems in a robust and flexible way.

[1] Email: grant@csc.liv.ac.uk

1571-0661 © 2006 Elsevier B.V. Open access under CC BY-NC-ND license.
doi:10.1016/j.entcs.2006.05.024

Yet it is widely agreed that the basic mechanism of interaction through method invocation does not meet the challenges posed by decentralised software systems. For example, Andrade and Fiadeiro [1], concerned with service-oriented software, describe (*op. cit.*, p. 380) 'a gap between the high-level specification of interactions and their implementation in any particular technology,' such as object-oriented languages that have a rigid coupling between methods and the (identities of the) objects that provide them. In a similar vein, Arbab [2], on component-based software, points to the 'tight coupling inherent in the method call semantics (sic) [which is] more appropriate for intra-component communication' (*op. cit.*, p. 8). Since object-orientation can be seen as based on the notion of abstract data type (ADT) by a correspondence between the operations of an ADT and the methods of a class (cf. Meyer [15]), Arbab writes: '[i]f a component, like an ADT, provides a set of operations, then the only way to communicate with a component is by *invoking* its operations, and inter-component communication becomes the same as inter-object communication' (*op. cit.*, p. 14). Moreover, '[c]omposition of two components, in such models, does not by itself yield another component' (*op. cit.*, p. 48). We agree with these arguments that the compositional techniques of the object paradigm are too inflexible and 'brittle' to meet the challenges of decentralised software. For us these arguments raised the question: to what extent are algebraic specifications tied to the notion of interaction solely through method invocation? In this paper we argue that it is possible to use some of the methodologies of algebraic specification in formulating component composition where interaction takes place through shared subcomponents, thus relaxing the tight coupling of method-call semantics.

In Section 3, we use *hidden algebra* as a language for specifying components. Hidden algebra was introduced by Goguen [7] to capture the notion of behaviour in systems that have state. In viewing components as 'black boxes' with state, our approach is similar to that of Arbab [2]; it is also similar to coalgebraic approaches such as that of Barbosa [3,4], since hidden algebra is closely related to a restricted form of coalgebra [13,5]. The approach also has the advantage that the possible implementations of a hidden specification are the models of the specification.

In hidden algebra, composite (distributed) systems can be built up from components by *concurrent connection*; this was introduced in [8] as a purely syntactic construct. One of the contributions of the present work is an examination of the semantics of this construct in terms of its model theory. In Section 4, we introduce the notion of *meromorphism* to capture the inclusion of one component within another, and show that meromorphisms make concurrent connection into a (co)limit construction: i.e., concurrent connection of component specifications is the 'least' way of combining components with meromorphisms from the components to the composite system. The other key contribution of the present work is to suggest that concurrent connection could usefully form the basis of a component composition language, by showing that the construction extends in a natural way to models. If we build a composite specification by means of concurrent connection, we should also require that models — i.e., implementations — of the individual components should be composable to give a model of the composite system. In Section 5 we

give 'amalgamation lemmas' (in the terminology of Ehrig and Mahr [6]) which show that this is indeed the case. A pleasant consequence of these results is that systems can be refined by refining individual components.

Before we set out the main results, Section 2 gives some background on algebraic specification and hidden algebra. Some of the technical development in this paper is couched in the language of category theory, because this allows for a very concise statement of key results. However, we have tried to make this as transparent as possible by paraphrasing and explaining the intuitions behind the categorical terminology. In particular, although many of these results would be most naturally expressed in terms of indexed categories, we have adopted a more elementary approach, particularly in the proofs, in order that the actual constructions be given explicitly.

2 Preliminaries

This section fixes terminology and notation that is used in the following sections. We assume familiarity with algebraic specification, and so do not motivate the notions described below. For an introduction to algebraic specification, see Meinke and Tucker [14]; for more background on hidden algebra, see Goguen and Malcolm [11].

A signature is a pair (S, Σ) consisting of a set S of sorts and an $(S^* \times S)$-indexed set Σ of operations. We usually denote such a signature as Σ if the set S can be left implicit, and we sometimes write $\sigma : w \to s$ for $\sigma \in \Sigma_{w,s}$. A Σ-*algebra* A interprets sorts as sets, and operations as appropriately typed functions; we write $A(s)$ for the carrier set of sort $s \in S$, and $A(\sigma) : A^w \to A_s$ for the function interpreting $\sigma : w \to s$. A signature morphism $(S, \Sigma) \to (S', \Sigma')$ is a pair (f, g), with $f : S \to S'$, and $g : \Sigma \to \Sigma'_{f^*,f}$ an $(S^* \times S)$-sorted function, i.e., for $w \in S^*$ and $s \in S$, $g_{w,s} : \Sigma_{w,s} \to \Sigma'_{f^*(w),f(s)}$. Usually, we denote a signature morphism by $\varphi : \Sigma \to \Sigma'$, and ignore the distinction between f and g and drop all subscripts, writing $\varphi(s)$ for $f(s)$, and $\varphi(\sigma)$ for $g_{w,s}(\sigma)$.

We write $T_\Sigma(X)$ for the algebra of Σ-terms with variables from the S-sorted set X. A theory is a pair (Σ, E), where Σ is a signature, and E a set of Σ-equations of the form $(\forall X)\ l = r$, where X is an S-sorted set of variables, and $l, r \in T_\Sigma(X)_s$ for some $s \in S$. A (Σ, E)-model is a Σ-algebra A that satisfies all the equations in E; we write $A \models E$ to indicate that A is a (Σ, E)-model.

Hidden algebra distinguishes *hidden* and *visible* sorts; hidden sorts are intended to represent states that can change, while visible sorts represent immutable data values, such as numbers, Booleans, etc. A hidden theory specifies behaviour of states, and uses a fixed representation of the visible sorts: a *visible data universe* is a triple (V, Ψ, D), where (V, Ψ) is a signature, and D is a Ψ-algebra. The examples in this paper use the visible data universe given by:

```
obj DATA is
  sorts Msg MsgList .
  op nil : -> MsgList .
  op cons : Msg MsgList -> MsgList .
  op isEmpty : MsgList -> Bool .
  var M : Msg .   var MS : MsgList .
  eq  isEmpty(nil)         =  true .
```

```
    eq  isEmpty(cons(M,MS))  =  false .
  end
```

(we use the notation of BOBJ [9], which we hope requires no explanation), and we assume some model D that satisfies the given equations.

Given a fixed visible data universe as above, a *hidden signature* is a pair (H, Σ), where the elements of H are called *hidden sorts* and are disjoint from V, and $(V \cup H, \Sigma)$ is a signature such that for $\sigma : w \to s$, there is at most one hidden sort in w (i.e., operations work locally on one state). Hidden Σ-algebras (or 'models') are Σ-algebras that agree with D on visible sorts and operations.

Example 2.1 *Two simple examples of hidden signatures, both of which extend the* `DATA` *signature, are:*

```
bth STATE is pr DATA .
   sort State .
end
```

```
bth ENLIST is pr STATE .
   op addToList : State Msg -> State .
end
```

A `STATE`*-algebra simply has some carrier set for states, in addition to the fixed interpretation of* `DATA`*. An* `ENLIST`*-algebra A has one function A*(`addToList`) : A(`State`)$\times D$(`Msg`) $\to A$(`State`).

Behaviour of operations is given by equations: a hidden theory (Σ, E) consists of a hidden signature Σ, and a set of $(\Psi \cup \Sigma)$-equations E, where each equation contains at most one variable of hidden sort. A *context* is a term with exactly one occurrence of a place-holder variable (say, '_'); visible contexts are contexts of visible sort. We write $C_\Sigma(s, t)$ for the set of contexts of sort t that contain a place-holder variable of sort s; we write $c[t]$ for the term obtained by substituting term t for the place-holder variable in context c.

A hidden algebra **behaviourally satisfies** an equation $(\forall X)\ l = r$ iff it satisfies all equations of the form $(\forall X)\ c[l] = c[r]$, where c is a visible context.

Example 2.2 *A model of the following stores a list of messages:*

```
bth SEELIST is pr ENLIST .
   op list : State -> MsgList .
   op empty : State -> State .
   var S : State .  var M : Msg .
   eq  list(addToList(S,M))  =  cons(M, list(S)) .
   eq  list(empty(S))        =  nil .
end
```

We will not be concerned in this paper with behavioural satisfaction, but we note that behavioural satisfaction of visible-sorted equations such as those above is the same as standard satisfaction of equations [11].

A hidden homomorphism $h : A \to B$ of Σ-algebras is a homomorphism which is the identity on visible sorts.

In this paper we will consider only hidden signatures with no *generalised constants*, i.e., operations $\sigma : w \to s$ where $w \in V^*$ and $s \in H$. Thus there are no 'constructors' for states. This restriction gives us *final algebras*: for a hidden signature Σ, define F_Σ by $F_\Sigma(h) = \prod_{v \in V}[C_\Sigma(h, v) \to D(v)]$, so that $F_\Sigma(h)$ consists of 'abstract states' that map any visible context $c \in C_\Sigma(h, v)$ to a 'result' in $D(v)$. The interpretation of operations on this Σ-algebra is the obvious one. This algebra is final in that there is exactly one hidden homomorphism

$A \to F_\Sigma$ for any Σ-algebra A; this homomorphism maps a state of A to its abstract behaviour. More generally, Cîrstea [5] has shown that any theory morphism $\varphi : (\Sigma, E) \to (\Sigma', E')$ allows (Σ, E)-models to be extended to (Σ', E')-models. Technically, the reduct functor $A' \mapsto A'|_\varphi : \mathsf{Mod}(\Sigma', E') \to \mathsf{Mod}(\Sigma, E)$ has a right adjoint $A \mapsto A^\varphi : \mathsf{Mod}(\Sigma, E) \to \mathsf{Mod}(\Sigma', E')$. Thus for every $A \models (\Sigma, E)$, there is $A^\varphi \models (\Sigma', E')$, with $\varepsilon_A : A^\varphi|_\varphi \to M$ such that for every $h : A'|_\varphi \to A$ there is a unique $f/\varepsilon_A : A' \to A^\varphi$ such that $(f/\varepsilon_A)|_\varphi ; \varepsilon_A = f$. To simplify only slightly, the carrier of A^φ consists of elements p in $F_{\Sigma'}$ which are 'decorated' with elements of A which agree with p with respect to Σ-contexts (and we restrict to only those p that satisfy the equations E'). For operations $\sigma \in \Sigma'/\Sigma$, an element p is mapped to, say, p', and there is no restriction on the elements of A that decorate p and p'.

3 Component Specifications

Our notion of a component is a 'black box' with hidden state:

Definition 3.1 A component specification is a hidden theory with exactly one hidden sort.

We will usually use '$\mathcal{C}$', with subscripts or other decorations, as a variable ranging over component specifications, with $\mathcal{C}^\Sigma$ representing its signature, and $\mathcal{C}^E$ it equations, and write $A \models \mathcal{C}$ to indicate that A is a $\mathcal{C}$-model. We will typically denote the unique hidden sort of an component specification by '**h**'. In specific examples in BOBJ notation, we use State as the name of the unique hidden sort, as in Example 2.1. Although we do not suggest that components are classes or objects, it is standard terminology in hidden algebra, which we shall follow here, to refer to operations that return a visible sort as 'attributes', and operations that return a hidden sort as 'methods'. In general, attributes and methods take one hidden-sorted argument and some number of visible-sorted arguments; henceforth, we shall ignore the visible-sorted arguments, without loss of generality, as we could assume we have one operation for each of its possible fixed arguments, for example, an operation $\texttt{addToList}_m : \mathbf{h} \to \mathbf{h}$ for each $m \in D(\texttt{Msg})$.

Example 3.2 *A channel that carries values of sort* Msg *and a component that keeps a running total of the messages sent on a channel is specified as follows:*

```
bth CHANNEL is pr STATE .
   op read : State -> State .
   op val : State -> Msg .
end

bth ADDER is pr CHANNEL .
   op total : State -> MsgList .
   var S : State .
   eq  total(read(S))  =  cons(val(S), total(S)) .
end
```

Note that '`pr CHANNEL`' imports the `CHANNEL` specification, nad makes the channel a subcomponent of `ADDER`.

4 Concurrent Connection

In this section we describe the construction of complex components by *concurrent connection* [8], and give a formal definition of subcomponent which makes this a closure operation: i.e., the connected components are subcomponents of the composite system. We begin with the simplest case from [8]:

Definition 4.1 Given component specifications $\mathcal{C}_i$ for $i \in I$, the **independent sum** $||_{i\in I}\mathcal{C}_i$ is the component specification whose signature is the union $\cup_{i\in I}\mathcal{C}_i^{\Sigma}$ (which we assume to be disjoint, with the exception of the hidden sort **h**), and whose equations are the union of the equations in each $\mathcal{C}_i$, together with **independence axioms** of the form

$$(\forall S : \mathbf{h})\ \alpha(\mu(S)) = \alpha(S)$$

for each attribute α in $\mathcal{C}_i$ and method μ in $\mathcal{C}_j$ with $i \neq j$.

This represents two components with no communication. For example, the independent sum of two `SEELIST` components (cf. Example 2.2) would represent two separate lists, with operations

```
ops list1 list2 : State -> MsgList .
ops addToList1 addToList2 : State Msg -> State .
```

and the independence axioms would include:

```
eq  list1(addToList2(S,M))  =  list1(S) .
```

The following generalises this to allow for components that communicate via a shared subcomponent (again, from [8]).

Definition 4.2 Given component specifications $\mathcal{C}_i$ for $i = 0, 1, 2$, and hidden theory morphisms $\varphi_i : \mathcal{C}_0 \to \mathcal{C}_i$ for $i = 1, 2$, the **concurrent connection** is given by the component specification $\varphi_1 \Downarrow \varphi_2$ together with $\psi_i : \mathcal{C}_i \to \varphi_1 \Downarrow \varphi_2$ where the signature $(\varphi_1 \Downarrow \varphi_2)^{\Sigma}$ (and the morphisms ψ_i) is given by the pushout of the morphisms φ_i, and the equations $\varphi_1 \Downarrow \varphi_2^E$ contain all the equations $\psi_i(E_i)$ for $i = 1, 2$, as well as the **independence axioms**:

- for each $i \neq j \in \{1, 2\}$ and operations $\alpha_i : \mathbf{h} \to v$ in $\mathcal{C}_i$ and not in the range of φ_i and $\mu_j : \mathbf{h} \to \mathbf{h}$ in $\mathcal{C}_j$ and not in the range of φ_j, an equation

$$(\forall S : \mathbf{h})\ \psi_i(\alpha_i)(\psi_j(\mu_j)(S)) = \psi_i(\alpha_i)(S)\ .$$

Example 4.3 *A component that writes messages to a channel is specified in the following theory. This component stores messages (in an attribute* `store`*), which are written to the channel itself by the channel's* `read` *method.*

```
bth SENDER is pr CHANNEL .
   op put : State Msg -> State .
   op store : State -> Msg .
   var S : State .    var M : Msg .
   eq  store(put(S,M))  =  M .
   eq  store(read(S))  =  store(S) .
   eq  val(put(S,M))  =  val(S) .
   eq  val(read(S))  =  store(S) .
end
```

The concurrent connection of this component with the `ADDER` *component of Example 3.2, for which we use the notation* `SENDER/CHANNEL\ADDER`*, has all the opera-*

tions from ADDER *and* SENDER *(without duplicating the* CHANNEL *operations), all the equations from these two components, plus the following independence axiom:*

```
eq  total(put(S,M))  =  total(S) .
```

stating that the method put*, which is local to the sender, has no effect on the attribute* total*, which is local to the adder.*

The signature of the concurrent connection is formed by taking all the operations from the signatures of the components; no duplicates are made of any operations that come from shared subcomponents, but if two operations 'accidentally' have the same name (i.e., they don't come from a shared subcomponent) then they are named apart. This is an example of a colimit construction: the signature of the concurrent connection is a maximal way of combining the signatures of components without duplicating operations from shared subcomponents. This maximality gives a *universal property* that we will make frequent use of: given another signature, say Σ, that combines the component signatures without duplication, there is a unique signature morphism from the signature of the concurrent connection to Σ that includes the operations in the same way they are combined in Σ. Our main results below extend this sort of 'maximality' property to component specifications.

Now we capture the notion of subcomponent as follows:

Definition 4.4 A **meromorphism** $\varphi : (\Sigma, E) \to (\Sigma', E')$ is an inclusion of component specifications such that for all attributes α in Σ and all methods μ in Σ'/Σ, we have $E' \models (\forall S : \mathbf{h})\ \alpha(\mu(S)) = \alpha(S)$. We say that σ is **local to** (Σ', E') iff $\sigma \in \Sigma'/\Sigma$.

Note that since φ is an inclusion, we simply write α for $\varphi(\alpha)$ in this definition, and in the rest of this paper.

Proposition 4.5 *Let $\varphi_i : \mathcal{C}_0 \to \mathcal{C}_i$ $(i = 1, 2)$ be meromorphisms; then the concurrent connection injections $\psi_i : \mathcal{C}_i \to \varphi_1 \Downarrow \varphi_2$ are meromorphisms.*

Proof. Let α be an attribute in $\mathcal{C}_1$ and μ be a local method in $\mathcal{C}_2$. Now if α is local in $\mathcal{C}_1$, i.e., if α is not in $\mathcal{C}_0$, then

$$(\varphi_1 \Downarrow \varphi_2)^E \models (\forall S : \mathbf{h})\ \alpha(\mu(S)) = \alpha(S)$$

because that is an independence axiom in $(\varphi_1 \Downarrow \varphi_2)^E$. If α is in $\mathcal{C}$, then $\mathcal{C}_2^E \models (\forall S : \mathbf{h})\ \alpha(\mu(S)) = \alpha(S)$ by the condition that φ_2 is a meromorphism, and so it follows that $E' \models \alpha(\mu(S)) = \alpha(S)$. This shows that φ_1 is a meromorphism; symmetry between $i = 1$ and $i = 2$ shows that φ_2 is also a meromorphism. □

Let HMer be the category of component specifications and meromorphisms.

Proposition 4.6 *The category* HMer *is cocomplete.*

Proof. The theory STATE is an initial object. It has no attributes, so the defining property of meromorphisms is vacuously satisfied by its inclusion in any hidden component specification. Proposition 4.7 below shows that HMer has coproducts, and Proposition 4.8 shows that it has pushouts. □

Proposition 4.7 *Independent sum gives coproducts in* HMer.

Proof. Let $\mathcal{C}_i$ be hidden component specifications (for i in some index set I), and let $\mathcal{C}$ be their independent sum. The independence axioms of the independent sum are exactly what is needed to make the injections $\psi_i : \mathcal{C}_i \to \mathcal{C}$ meromorphisms. Now suppose $\chi_i : \mathcal{C}_i \to \mathcal{C}'$ is a cocone of meromorphisms. The universal property of $\mathcal{C}^\Sigma$ gives a unique $\xi : \mathcal{C}^\Sigma \to \mathcal{C}^\Sigma$ such that $\psi_i ; \xi = \chi_i$. To see that this is a meromorphism, let α be an attribute in $\mathcal{C}$, and let μ be local to $\mathcal{C}'$. Since α is in $\mathcal{C}$, it must be in $\mathcal{C}_i$ for some $i \in I$, and so $\mathcal{C}'^E \models \alpha(\mu(S)) = \alpha(S)$ because χ_i is a meromorphism; but this is exactly what is required for ξ to be a meromorphism. □

Proposition 4.8 *Concurrent connection is a pushout in* HMer.

Proof. Let $\psi_i : \mathcal{C}_i \to \mathcal{C}$ be the concurrent connection of $\varphi_i : \mathcal{C}_0 \to \mathcal{C}_i$ $(i = 1, 2)$, and let $\chi_i : \mathcal{C}_i \to \mathcal{C}'$ be a cocone of meromorphisms. By cocompleteness of the category of signatures, there is a unique morphism of cocones $\xi : \mathcal{C} \to \mathcal{C}'$; the proof that this is a meromorphism is similar to that in Proposition 4.7 above. □

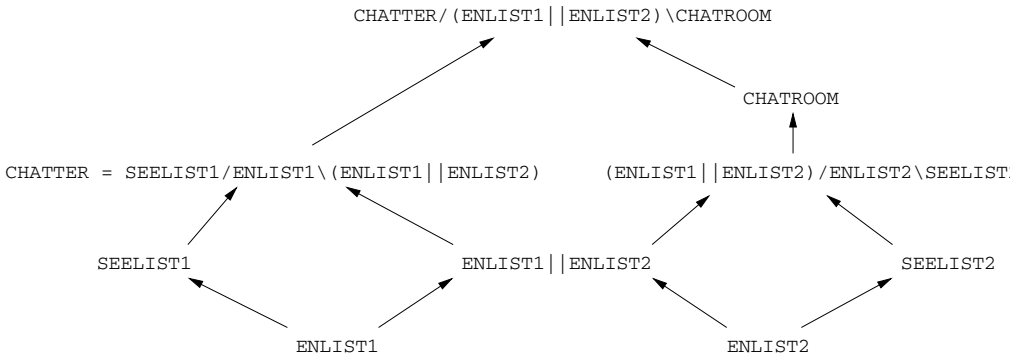
The point of these latter two propositions is that all colimits can be built from repeated construction of independent sums and concurrent connections. For example, consider a chatroom application, where a chatter can send messages to a chatroom, and the chatroom broadcasts messages to the chatter. If we concentrate only on the 'in-boxes' of both the chatter and the chatroom, we can specify the chatter as:

```
bth CHATTER is
   pr ENLIST *(op addToList to sendMsg) .
   pr SEELIST *(op addToList to getMsg , op list to history,
                op empty to clear) .
end
```

This component has two lists (with appropriately renamed operations), representing the ('write-only') list of messages the chatter sends to the chatroom, and the ('read-write') list of messages received from the chatroom. Similarly, the chatroom can be specified as:

```
bth CHATROOM is
  pr ENLIST *(op addToList to sendMsg) .
  pr SEELIST *(op addToList to receiveMsg , op list to inbox,
               op empty to broadcast) .
  op sendAll : State MsgList -> State .
  var S : State .  var M : Msg .   var MS : MsgList .
  eq  broadcast(S)  =  sendAll(S, inbox(S)) .
  eq  sendAll(S, empty)  =  S .
  eq  sendAll(S, cons(M,MS))  =  sendAll(sendMsg(S,M), MS) .
end
```

Now this is all we need specify; these two components share two lists with differing 'read-permissions', and the component representing their composition can be built up incrementally as in the following diagram.



This process could be repeated to form a component comprising two or more chatters sharing a chatroom.

5 Composing Implementations

We turn now to the question of composing models, i.e., implementations, of component specifications.

Definition 5.1 Let CMod be the category whose objects are pairs $(\mathcal{C}, M)$, where $\mathcal{C}$ is a component specification, and M a hidden $\mathcal{C}$-model, and whose arrows are pairs $(\varphi, h) : (\mathcal{C}, M) \to (\mathcal{C}', M')$ with $\varphi : \mathcal{C} \to \mathcal{C}'$ a meromorphism and $h : M'|_\varphi \to M$ a hidden homomorphism. Let CModSL be the subcategory of CMod where the morphisms satisfy the **strong localisation** property: that h is such that $h(M'_\mu(x)) = h(x)$ for all local methods μ in $\mathcal{C}'$.

The following 'cocompleteness' results are amalgamation properties: implementations of components give rise to implementations of specifications built from independent sum and concurrent connection. We begin with the simplest case, where the strong localisation property states that local methods have no effect at all on subcomponents. Note that this property holds for systems built exclusively by independent sums.

Proposition 5.2 CModSL *is cocomplete.*

Proof. We note that an initial object is given by $(\mathsf{State}, \mathbb{1})$, where $\mathbb{1}(\mathbf{h}) = \{0\}$. The following lemmas show the existence of coproducts and pushouts. □

Lemma 5.3 CModSL *has coproducts.*

Proof. Given a collection $(\mathcal{C}_i, M_i)$ for $i \in I$, let $\varphi_i : \mathcal{C}_i \to ||_{i\in I}\mathcal{C}_i$ be the independent sum, and define the model $M \models ||_{i\in I}\mathcal{C}_i$ by

$$M(\mathbf{h}) = \prod_{i\in I} M_i(\mathbf{h})$$

For σ local to $\mathcal{C}_j^\Sigma$, $M(\sigma)$ takes a family $(m_i)_{i\in I}$ to $(m'_i)_{i\in I}$, where $m'_i = m_i$ for all $i \neq j$, and $m'_j = M_i(\sigma)(m_j)$. It is straightforward to see that M satisfies the equations of $||_{i\in I}\mathcal{C}_i$. Then $(\varphi_i, \pi_i) : (\mathcal{C}_i, M_i) \to (||_{i\in I}\mathcal{C}_i, M)$ for $i \in I$, where

$\pi_i : M\restriction_{\varphi_i} \to M_i$ is the obvious projection. It is clear that the projections satisfy the strong localisation property. Given a cocone $(\psi_i, h_i) : (\mathcal{C}_i, M_i) \to (\mathcal{C}', M')$, the universal property of $||_{i\in I}\mathcal{C}_i$ gives $\chi : ||_{i\in I}\mathcal{C}_i \to \mathcal{C}'$, and since $M(\mathbf{h})$ is a product, we have $h/\pi : \chi M' \to M$ defined by $h/\pi\,;\pi_i = h_i$ for all $i \in I$. It follows almost immediately from this definition that h/π has the strong localisation property if each h_i does. To see that this is a homomorphism for the signature of the independent sum, let σ be local to $\mathcal{C}_j$, then

$$
\begin{array}{ll}
 & M'\restriction_\chi(\sigma)\,;h/\pi\,;\pi_j \\
= & \\
 & M'(\sigma)\,;h_j \\
= & \\
 & h_j\,;M_j(\sigma) \\
= & \\
 & h/\pi\,;\pi_j\,;M_j(\sigma) \\
= & \\
 & h/\pi\,;M(\sigma)\,;\pi_j
\end{array}
$$

and for $i \neq j$,

$$
\begin{array}{ll}
 & M'\restriction_\chi(\sigma)\,;h/\pi\,;\pi_i \\
= & \\
 & M'(\sigma)\,;h_i \\
= & \qquad \{\ h_i \text{ has the strong localisation property } \} \\
 & h_i \\
= & \\
 & h/\pi\,;\pi_i \\
= & \\
 & h/\pi\,;M(\sigma)\,;\pi_i
\end{array}
$$

and so $M'\restriction_\chi(\sigma)\,;h/\pi = h/\pi\,;M(\sigma)$. □

Lemma 5.4 CModSL *has pushouts.*

Proof. Given a span $(\varphi_i, h_i) : (\mathcal{C}_0, M_0) \to (\mathcal{C}_i, M_i)$ $(i = 1, 2)$ in CModSL define the model M of the concurrent connection $\varphi_1 \Downarrow \varphi_2$ by

- $M(\mathbf{h}) = \{\ (m_1, m_2) \in M_1(\mathbf{h}) \times M_2(\mathbf{h}) \ \mid\ h_1(m_1) = h_2(m_2)\ \}$
- $M(\mu)(m_1, m_2) = (M_1(\mu)(m_1), M_2(\mu)(m_2))$ for μ in $\mathcal{C}$
- $M(\mu)(m_1, m_2) = (M_1(\mu)(m_1), m_2)$ for μ local to $\mathcal{C}_1$
- $M(\mu)(m_1, m_2) = (m_1, M_2(\mu)(m_2))$ for μ local to $\mathcal{C}_2$
- $M(\alpha)(m_1, m_2) = M_1(\alpha)(m_1) = M_2(\alpha)(m_2)$ for α in $\mathcal{C}$
- $M(\alpha)(m_1, m_2) = M_i(\alpha)(m_i)$ for α local to $\mathcal{C}_i$ $(i = 1, 2)$.

Note that the second-last bullet point is well defined since each h_i is the identity on visible sorts, hence $M_1(\alpha)(m_1) = h_1(M_1(\alpha)(m_1)) = M_0(\alpha)(h_1(m_1)) = M_0(\alpha)(h_2(m_2)) = h_2(M_2(\alpha)(m_2)) = M_2(\alpha)(m_2)$. Letting $\psi_i : \mathcal{C}_i \to \varphi_1 \Downarrow \varphi_2$ denote the inclusions into the concurrent connection, we have the obvious projections $\pi_i : M\restriction_{\psi_i} \to M_i$, and it is quite straightforward to see that these are indeed homomorphisms. It is straightforward to show that $M \models E_i$, so it only remains to show that M satisfies the independence axioms. Let α be local to $\mathcal{C}_1$ and μ be local to

$\mathcal{C}_2$; for any $(m_1, m_2) \in M(\mathbf{h})$

$$\begin{array}{ll} & M(\alpha)(M(\mu)(m_1, m_2)) \\ = & \\ & M(\alpha)(m_1, M_2(\mu)(m_2)) \\ = & \\ & M_1(\alpha)(m_1) \\ = & \\ & M(\alpha)(m_1, m_2) \end{array}$$

whence (and by symmetry between subscripts 1 and 2), $M \models (\varphi_1 \Downarrow \varphi_2)^E$.

Now, if $(\chi_i, g_i) : (\mathcal{C}_i, M_i) \to (\mathcal{C}', M')$ is a cocone for (φ_i, h_i), then we have $\xi : (\varphi_1 \Downarrow \varphi_2)^\Sigma \to \mathcal{C}'^\Sigma$ with $\varphi_i ; \xi = \psi_i$. Define $f : \xi\restriction_{M'} \to M$ by $f(m') = (g_1(m'), g_2(m'))$, which is well defined since $g_1\restriction_{\psi_1} ; h_1 = g_2\restriction_{\psi_2} ; h_2$. To see that this is a homomorphism, consider an operation σ in $(\varphi_1 \Downarrow \varphi_2)^\Sigma$. If σ is in $\mathcal{C}_0^\Sigma$, then $M(\sigma)(f(m')) = M(\sigma)(g_1(m'), g_2(m')) = (M_1(\sigma)(g_1(m')), M_2(\sigma)(g_2(m'))) = (g_1(M'(\sigma)(m')), g_2(M'(\sigma)(m'))) = f(M'(\sigma)(m'))$. Otherwise, σ is local, say to $\mathcal{C}_1^\Sigma$, in which case

$$\begin{array}{ll} & M(\sigma)(f(m')) \\ = & \\ & M(\sigma)(g_1(m'), g_2(m')) \\ = & \\ & (M_1(\sigma)(g_1(m')), g_2(m')) \\ = & \\ & (g_1(M'(\sigma)(m')), g_2(m')) \\ = & \quad \{ \text{ strong localisation property of } g_2 \ \} \\ & (g_1(M'(\sigma)(m')), g_2(M'(\sigma)(m'))) \\ = & \\ & f(M'(\sigma)(m')) \end{array}$$

The case where σ is local to $\mathcal{C}_2^\Sigma$ is symmetric, so f is indeed a homomorphism, and it is clear that it is the unique f such that $f\restriction_{\psi_i} ; \pi_i = g_i$. □

The strong localisation property of CModSL is indeed a strong restriction on possible communication between subcomponents: it states that local methods can have no effect whatsoever on shared subcomponents, and so it does not apply to the SENDER component of Example 4.3. So we turn our attention now to CMod, we will see that the cofree extensions of Cîrstea [5] described in the Preliminaries gives a more powerful amalgamation result.

Lemma 5.5 CMod *has pushouts.*

Proof. Suppose $\mathcal{C}_i$ $(i = 0, 1, 2)$ are component specifications, with meromorphisms $\varphi_i : \mathcal{C}_0 \to \mathcal{C}_i$ $(i = 1, 2)$, and models $M_i \models \mathcal{C}_i$ $(i = 0, 1, 2)$ with homomorphisms $h_i : M_i\restriction_{\varphi_i} \to M_0$ $(i = 1, 2)$. By cofree extension along $\varphi_i ; \psi_i$ (which we simply write as $\varphi ; \psi$), we have $\widehat{h_i} = (\varepsilon_{M_i}\restriction_{\varphi_i} ; h_i)/\varepsilon_{M_0} : M_i^{\psi_i} \to M_0^{\varphi ; \psi}$. Now let $g_i : M \to M_i^{\psi_i}$ be the pullback of these $\widehat{h_i}$, so we have $g_i\restriction_{\psi_i} ; \varepsilon_{M_i} : M\restriction_{\psi_i} \to M_i$. Note that

$$M(\mathbf{h}) = \{ \ (m_1, m_2) \in M_1^{\psi_1}(\mathbf{h}) \times M_2^{\psi_2}(\mathbf{h}) \ \mid \ \widehat{h_1}(m_1) = \widehat{h_2}(m_2) \ \}$$

By construction, $M \models \varphi_1 \Downarrow \varphi_2$, and the g_i are $(\varphi_1 \Downarrow \varphi_2)$-homomorphisms, so the

$g_i\restriction_{\psi_i} ; \varepsilon_{M_i}$ are $\mathcal{C}_i$-homomorphisms.

Now suppose $(\chi_i, f_i) : (\mathcal{C}_i, M_i) \to (\mathcal{C}', M')$ is a cocone. We have $f_i/\varepsilon_{M_i} : M'\restriction_\xi \to M_i^{\psi_i}$. Now

$$\begin{array}{ll} & (f_i/\varepsilon_{M_i} ; (\varepsilon_{M_i}\restriction_{\varphi_i} ; h_i)/\varepsilon_{M_0})\restriction_{\varphi\,;\,\psi} ; \varepsilon_{M_0} \\ = & \\ & (f_i/\varepsilon_{M_i})\restriction_{\varphi\,;\,\psi} ; ((\varepsilon_{M_i}\restriction_{\varphi_i} ; h_i)/\varepsilon_{M_0})\restriction_{\varphi\,;\,\psi} ; \varepsilon_{M_0} \\ = & \\ & (f_i/\varepsilon_{M_i})\restriction_{\varphi\,;\,\psi} ; \varepsilon_{M_i}\restriction_{\varphi_i} ; h_i \\ = & \\ & ((f_i/\varepsilon_{M_i})\restriction_{\psi_i} ; \varepsilon_{M_i})\restriction_{\varphi_i} ; h_i \\ = & \\ & f_i\restriction_{\varphi_i} ; h_i \end{array}$$

which shows that $f_i/\varepsilon_{M_i} ; \widehat{h_i} = (f_i\restriction_{\varphi_i} ; h_i)/\varepsilon_{M_0}$, and since $f_1\restriction_{\varphi_1} ; h_1 = f_2\restriction_{\varphi_2} ; h_2$, we have $f_1/\varepsilon_{M_1} ; \widehat{h_1} = f_2/\varepsilon_{M_2} ; \widehat{h_2}$, and so, by the universal property of M as a pullback, we have $(f/\varepsilon_M)/g : M'\restriction_\xi \to M$, i.e.,

$$(\xi, (f/\varepsilon_M)/g) : (\varphi_1 \Downarrow \varphi_2, M) \to (\mathcal{C}', M') \ .$$

For any $e : M'\restriction_\xi \to M$, we have

$$\begin{array}{ll} & e\restriction_{\psi_i} ; g_i\restriction_{\psi_i} ; \varepsilon_{M_i} = f_i \\ \text{iff} & \\ & (e\,;g_i)\restriction_{\psi_i} ; \varepsilon_{M_i} = f_i \\ \text{iff} & \\ & e\,;g_i = f_i/\varepsilon_{M_i} \\ \text{iff} & \\ & e = (f/\varepsilon_M)/g \end{array}$$

which gives the desired universal property. □

It is easily seen that $(\mathsf{State}, \mathbb{1})$ is also initial in CMod; thus, taking $(\mathcal{C}_0, M_0)$ in the lemma above to be $(\mathsf{State}, \mathbb{1})$ gives a way of constructing an amalgam $M_1||M_2$ of the independent sum $\mathcal{C}_1||\mathcal{C}_2$, which gives binary coproducts in CMod, and this can be generalised very straightforwardly to arbitrary coproducts, so we have our main

Theorem 5.6 (Amalgamation) CMod *is cocomplete.*

6 Conclusions

We have shown that hidden algebraic component specifications support component composition by concurrent connection, and that this composition extends naturally to composition of models, which we think of as concrete implementations of component specifications. We would argue that this suggests that hidden algebra could form a useful basis for component composition languages, though more technical work needs to be done, and, in particular, more examples and case studies would be needed to make this argument more persuasive.

Returning to the issues raised in the introduction, our main results show that it is possible to have a component model that is based on algebraic specifications and ADTs, and in which the result of composition is in itself a component. However, it

is an essentially algebraic approach, and therefore, like the coalgebraic approaches to which hidden algebra is related, it is based on operations. The objection could be raised that the approach is essentially object-oriented, and that our components are no more than the 'fortified collections of objects' referred to by Arbab [2]. But note that the model we propose incorporates communication between components through shared subcomponents, not merely through method invocation. We would argue that the use of operations in specification simply represents the possibilities for components to change state, and these do not represent methods, though of course the operations could be realised by actual methods in an object-oriented implementation. This more general view of operations can be illustrated by pointing out that they simply represent possible changes of state at particular levels of abstraction, and that the very flexibility of an algebraic approach allows one to move readily between different levels of abstraction. One consequence of our model-theoretic study of concurrent connection is that composite systems can be refined by refining individual components. Indeed, standard algebraic techniques for proving correctness of refinements, as well as techniques specific to hidden algebra [10], combine elegantly with the colimit constructions given above. As a specific example, the 'adding-to-a-list' component in Example 2.1 could be refined (moving down to a lower level of abstraction) by the composite presented in Example 4.3, which in turn could be refined by a more concrete specification of streams.

In order to develop our proposed approach, we need to extend the amalgamation results presented here to provide a uniform treatment of 'gluing' code, along the lines currently achieved by the language Reo [2]. One possibility would be to examine amalgamation results that combine individual states of models of components. Another avenue for future research is suggested by the chatroom example described at the end of Section 4: how to allow for dynamic reconfiguration of components, e.g., chatters joining and leaving the chatroom. The amalgamation properties presented in this paper could be useful here, perhaps in combination with rewriting logic and tile logic [12]).

References

[1] Andrade, L. F. and J. L. Fiadeiro, *Composition contracts for service interaction*, Journal of Universal Computer Science **10** (2004), pp. 375–390.

[2] Arbab, F., *Abstract Behavior Types: a foundation model for components and their composition*, Science of Computer Programming **55** (2005), pp. 3–52.

[3] Barbosa, L., *Components as processes: An exercise in coalgebraic modeling*, in: S. Smith and C. Talcott, editors, *Proc. FMOODS 2000* (2000), pp. 397–417.

[4] Barbosa, L., *Towards a calculus of software components*, Journal of Universal Computer Science **9** (2003), pp. 891–909.

[5] Cîrstea, C., *Coalgebra semantics for hidden algebra: parameterized objects and inheritance*, in: F. Parisi-Presicce, editor, *Proc. 12th Workshop on Algebraic Development Techniques* (1998), pp. 174–189.

[6] Ehrig, H. and B. Mahr, "Fundamentals of Algebraic Specification 1: Equations and Initial Semantics," Springer, 1985.

[7] Goguen, J. A., *Types as theories*, in: G. M. Reed, A. W. Roscoe and R. F. Wachter, editors, *Topology and Category Theory in Computer Science*, Oxford University Press, 1991 pp. 357–390.

[8] Goguen, J. A. and R. Diaconescu, *Towards an algebraic semantics for the object paradigm*, in: H. Ehrig and F. Orejas, editors, *Recent Trends in Data Type Specification* (1994), pp. 1–29.

[9] Goguen, J. A., K. Lin and G. Roşu, *Circular coinductive rewriting*, in: *Proceedings, Automated Software Engineering '00* (2000), pp. 123–131.

[10] Goguen, J. A. and G. Malcolm, *Proof of correctness of object representations*, in: A. W. Roscoe, editor, *A Classical Mind: essays dedicated to C.A.R. Hoare*, Prentice-Hall International, 1994 pp. 119–142.

[11] Goguen, J. A. and G. Malcolm, *A hidden agenda*, Theoretical Computer Science **245** (2000), pp. 55–101.

[12] Hirsch, D. and U. Montanari, *Consistent transformations for software architecture styles of distributed systems*, Electronic Notes in Theoretical Computer Science **28** (1999).

[13] Malcolm, G., *Behavioural equivalence, bisimulation, and minimal realisation*, in: M. Haveraaen, O. Owe and O.-J. Dahl, editors, *Recent Trends in Data Type Specifications* (1996), pp. 359–378.

[14] Meinke, K. and J. V. Tucker, *Universal algebra*, in: S. Abramsky, D. Gabbay, T. Maibaum, editors, *Handbook of Logic in Computer Science*, Vol. **1**, Oxford University Press, 1993 pp. 189–411.

[15] Meyer, B., "Object-Oriented Software Construction," Prentice Hall, 1997, 2nd edition.

[16] Resnick, M., "Turtles, Termites and Traffic Jams: explorations in massively parallel microworlds," MIT Press, 1994.